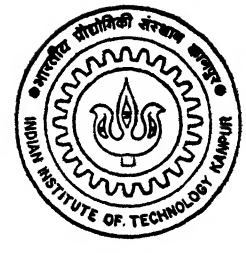
Translating Message Centric OO Specification to a RDBMS

by

M. Vijayanand



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

March, 1996

Translating Message Centric OO Specification to a RDBMS.

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

M. VIJAYANAND

to the

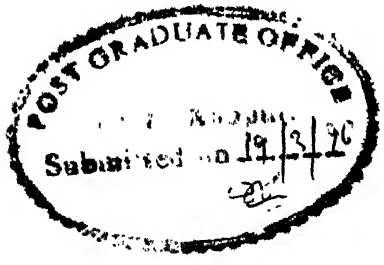
DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

March 1996.

CSE-1996-M-VIJ-TRA

CERTIFICATE

This is to certify that the work contained in the thesis entitled *"Translating Message Centric OO Specification to a RDBMS"* by *"M.Vijayanand"*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



Dr. Harish Karnick
Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

March 1996

Acknowledgments

I am deeply indebted to my guide, Prof. H. Karnick, for his consistent and friendly guidance throughout the thesis. This thesis would not have been completed so early had he not worked in the odd hours and at odd places. Special thanks for my group, Rafee and Sarma, for their coordination during this work. Thanks to mtech' 94 for giving a memorable company. Special thanks to my parents, brothers, sister and friends, outside here, for their encouragement and support throughout my stay here. Special thanks espicially to Davi and Panku for their help in the thesis. I would like to thank to all those who played Volley Ball and Badminton with me in the evenings which helped to recreate myself. I appreciate Raghu and Bera for their fighting nature in the games.

Finally I would like to thank the institute.

Abstract

This thesis develops a translator, consisting of mapping rules, for constructs in the specification language, "A Message Based Specification System for Object Oriented Software Construction" developed previously [Sar96], to constructs of a RDBMS (in this case ORACLE) This language being dealt basically has two models, the object model and the message model. Object model gives the static behavior of the system, i.e. the objects and their relationships among them. Message model gives the dynamic aspect of the system. In the process of translation ambiguities and other errors in the specification are checked for and reported. Both the static (i.e. the object model) and dynamic (message model) parts are translated. The message model is translated into PL/SQL templates.

Contents

List of Figures

Chapter 1

Motivation and Introduction.

1.1 Introduction:

To construct software, generally, two approaches are followed. One is a structured design approach and the other is an object oriented approach. The first one mainly models the application in terms of functions, hence it is also called a functional methodology. The latter one models it in terms of objects and relationships among them.

Though which approach is better, is mostly application dependent, both approaches are being actively researched. Both these approaches have several prescriptions in the form of methodologies which lay down guidelines about what should be done when going through the various stages of software construction.

In Object Oriented approaches some of the well known methodologies are due to Coad-Yourdon, Booch, Jacobson(Object Oriented Software Engineering), Rumbaugh(Object Modeling Technique).

Most of these methodologies represent the dynamic nature of the system using state diagrams and data flow diagrams. The state diagrams do not support representation for concurrency. The data flow diagrams hide the state of the object.

In his M.Tech thesis, *A Message Based Specification System for Object Oriented Software Construction*, Sarma [Sar96], has attempted to develop a new methodology which presents a notation to capture more completely the dynamic aspect of an

application. This methodology is based on a message passing paradigm and is represented in a language invented for this purpose. Specifications in this methodology are written in two parts. The first part gives the static nature of the problem, ie defining classes and relationships among them. The second part deals with the dynamic nature of the the objects. The dynamic nature, ie the behavior of the objects is represented by the messages, these objects receive and send and seen to capture the functional, dynamic and process oriented aspects of the application. The notation tries to make it easy for an analyser/designer to represent the requirements. The representation also seeks to make tool building easier so that design can be readily translated to implementations.

As a part of this attempt, this thesis attempts to develop a tool which maps as many constructs as possible to an RDBMS platform which is not an object oriented language. In parallel, work is in progress to map this into the object oriented language, C++.

1.1.1 Overview of the report

Chapter 2, describes the notation for the methodology. Chapter 3 deals with the design of the translator. Mapping rules with some examples are given in chapter 4. Limitations, related work and Conclusions are given in the last chapter.

1.2 Motivation

Object oriented approaches have been found useful in the design and construction of software.

Databases provides inherent persistence. DBMS also gives different views to the database for different applications. Data can be shared across different applications. This is because a DBMS provides atomicity, persistence and concurrency. Another reason for choosing an existing relational database platform is the maturity and stability that existing database systems have achieved. Relational databases are used instead of hierarchical or network databases, because Relational databases

have been found to be more efficient, flexible and commercially successful than their counter parts.

Relational databases are used instead of object oriented databases, because object oriented databases are still not as widely available neither are they as stable. Besides there are no standards for OODBMSs.

This thesis attempts to map a specification written using the notation given, to an RDBMS platform. The output produced will be helpful in writing the code. This automatic generation of templates would be helpful as manually translating would be time consuming, tedious and error prone.

Chapter 2

Message Based Specification Language

In this Chapter we introduce the message based specification language and give a detailed description of the notation. The notation is specified in appendix C.

2.1 Concepts of the Method

In the method the system is viewed as a collection of objects communicating with each other. Each object has some data that constitute its state and some rules that tell when to send a message, when to receive a message and what to do when a message is received . As all these rules involve messages, the rules are specified with the message rather than with the object.

Below, some basic terms that are used in this method are introduced. This may be more clearly understood from [Sar96]

Object An Object is an abstract or real world entity which has state, behavior and identity, and communicates with other objects.

State The state of an object at an instant consists of its knowledge about itself and its knowledge about the environment at that instant.

message A message forms the unit of communication between two objects.

The message is defined by the sender, receiver, message identifier and the arguments for the message. Also specified along with the message are Pre conditions (Constraints on the state of the sender to send the message), Post conditions (Constraints on the state of the receiver to receive the message) and Action to be performed by the receiver on receiving that message.

Process A Process is a sequence of events that happen to make a meaningful real-world functionality.

A Process can be viewed as a thread of activities. For example, in the DOAA Office automation system, Registration process starts with DOAA sending Registration notification and ends with Student's Roll Number being entered in the Rolls.

Generic process and Generic message Generic process is the mechanism by which we can represent similar turn of events once and reuse it later. Similarly generic messages can be used when similar messages are being sent to different objects.

Sub Process Sub process specification is useful to handle complexity. Large processes can be decomposed into sub processes. Generally if there is one central object that interacts with several objects to do an operation then we put these interactions into a sub process.

Chapter 3

Design

The specification written using the notation specified is parsed using **LEX** and **YACC**. No particular order in parsing the input is assumed, ie. one can give the input in any order. Input can be given without specifying classes, ie only processes may be given. Similarly, one can omit process and parse only classes. But, when dealing with process one cannot give only generic process or/and *sub process.* Processes have to start with *process* which gives the real meaning to the dynamic nature of the objects. *Sub process and generic sub process* are like functions in general procedural language. But *process* is like main. So *sub process and generic sub process* do not really mean any thing with out *process.* BNF notation of the grammar is given in APPENDIX A.

The parsed input is stored in a convenient form. This work used dynamic memory for this purpose, wherever possible. Separate linked lists are used to store *classes, processes, sub processes, generic processes* and *generic messages.* Also, separate lists have been maintained where needed. ie whenever temporary memory of small size is required such as 100bytes, static memory is used. For example, to store classes a list is maintained, to store the attributes of classes another list is maintained. Similarly, for others also separate lists are used.

In writing the specification, using the concepts given, one need not worry about sequencing or control flow. This is because events occur asynchronously. Sequencing of the messages has to be handled. For example, if the action for a message is to

send another message, the latter message, in the specification need not be written immediately following this message. This can be written any where in that process, sub process or generic sub process. If a message is not written in proper sequence messages are shuffled in such a way that the mesg_id in the "action" of a message will be same as the mesg_id of the next message. Since every action will have id of a message, a subprocess, generic sub process or a generic message, sorting can be done based on the id of the message. In sorting, only simple actions, ie which involve sending of ordinary messages are only considered. In other words, actions which do not involve doing sub process, generic sub process or sending generic message are only considered in sorting. This is because, other actions like doing a sub process, doing generic sub process or sending generic message are handled in a different fashion.

Example:

For the specification:

FROM : DOAA_INTERFACE.
TO : DOAA.
msg_id : send_reg_notice_to_depts(reg_notice < $NOTICE$ >
Action : se nd message notice_for_registration.
⋮
FROM : DOAA.
TO : CONVENER.
msg_id : notice_for_registration(reg_notice < $NOTICE$ >.
Action : do sub_process store_notice.

After sorting the order of the messages would be:

FROM : DOAA_INTERFACE.
TO : DOAA.
msg_id : send_reg_notice_to_depts(reg_notice < $NOTICE$ >
Action : send message notice_for_registration.

FROM : DOAA.
TO : CONVENER.

msg_id : notice_for_registration(reg_notice < $NOTICE$ >.
Action : do sub_process store_notice.

⋮

While dealing with actions which must do some sub process, generic sub process and send a generic message, except for sub process store, retrieve, select, control is transferred to the corresponding sub process, generic sub process or generic message. If in a message, the action requires a sub process, generic sub process or the sending of a generic message, control of mapping the specification is transferred from this message to the corresponding sub process, generic sub process or generic message where mapping of these subprocesses is done. After these are handled, control is restored to the the message from where it has come. From these sub processes you may have an action which deals again with some other subprocess. This is similar to functions and procedures in general procedural languages. Examples are given in the next chapter under the sub section "Action".

For a generic sub process or a generic message, constraints are processed before control is transferred to these sub processes.

After messages are sorted possible semantic errors in the *classes and messages* are traced. Details with examples, for errors are given in APPENDIX B.

After tracing of errors in classes and messages is done, views are created. Views are created for the *tables and lists* which occur as parameters in the message ids and results.

The overall design of the work is better understood from the fig. 3.1.

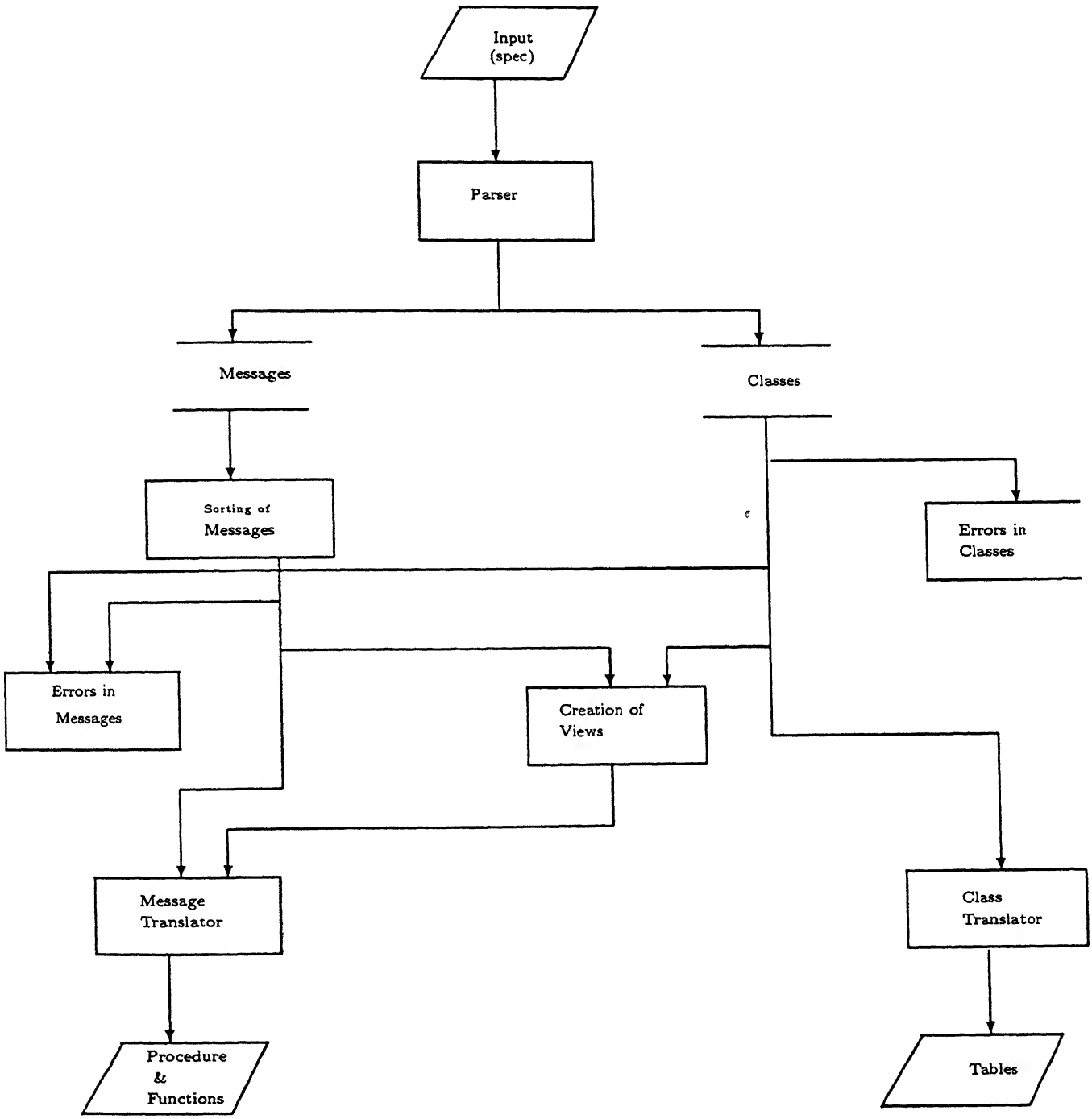


Figure 3.1: Design of the work

Chapter 4

Mapping Rules

In this chapter we discuss the, mapping of the constructs to *RDBMS* is given in this chapter. Since *RDBMS* is not a object oriented language, designing a Relational database is not easy. Also since *RDBMS* has many limitations in the data structures, mapping all the constructs is not possible. Mapping of *structures, tables and lists* is not direct. We have mapped these constructs also into tables and illustrate this with examples. *Messages* are mapped to *procedures* or *functions* in *ORACLE* which uses PL/SQL.

4.1 Classes

In this section the static nature of the specification ie classes, their attributes and the relationships among the classes are treated. Since structurally only tables are available in *RDBMS*, many of the static constructs are mapped to tables. Apart from ordinary data types like *string, number, etc*, there are some complex data structures like *lists, structures and tables.* All the relationships are mapped into tables which is the only way to store information in a *RDBMS.*

4.1.1 CLASS

CLASS is mapped to a table. The Class name becomes the table name. The attributes of the class are mapped to columns of the table. So, each instance of the

class will be considered as a row in the table.
If the type of the attribute is *STRING* or *string* then it is mapped to *VARCHAR(n)* which is the one generally used in *ORACLE* to represent a variable length character of length less than 2000 characters.

If the attribute is of type *NUMBER* it remains the same except that dimension of the number has to be given.

If attribute is of type *text* it is mapped to type *LONG* in *ORACLE*. With this data type one can give a text size up to 2 giga bytes.

If the attribute's type is *CHARACTER* which is used if the valid value is a single character, then it is mapped to type *CHAR*.For this type if no dimension is given , it is taken as 1.

If the type is of *DATE* or *YEAR* it is conveniently mapped to *DATE*, from which year can be retrieved.

Apart from all the attributes getting mapped to columns, some additional columns are also created. One such column is the object's id attribute. The other column could be the "table name", which will be there if the attributes contain structures, tables, or lists. Some times object ids of other related objects may also be included for the reasons given later in this chapter.

Despite the fact that a *RDBMS* does not provide inherent support for object ids, they are being used in this work because, databases are used via programs. So, accessing of the information would be easy and clear. The use of the ids is observed later in this chapter.

Also for each class we create a scratch table with only one column, to store the object id of the sender of any message to this class. The attribute is of type NUMBER. So if the class name is X, then a scratch table with the name X_scratch is created. This table would be helpful in replying. The use of this table can be observed clearly while dealing with the message specification of the problem.

Example:

For the specification:

CLASS : PERSON.

ATTR : name < *STRING* >, email_addr < *STRING* >,

address < *STRING* >.

The generated ORACLE code would be:

```
create table PERSON (
        PERSON_id       NUMBER
        name            VARCHAR(n)
        email_addr      VARCHAR(n)
        address         VARCHAR(n)
);
create table PERSON_scratch (
        id              NUMBER
);
```

This type of table is not shown in the examples.

The attributes of a class may include structures, tables or lists. All these are mapped into tables. In each of these tables, apart from the members in the corresponding structures, table or list as attributes, there will be an additional attribute which is the id of the parent class. In the corresponding parent class there will be an additional attribute which gives the table name of the structure or table or list. This type of additional information helps in accessing the information easily, ie navigating from one table to the other.

Example:

For the specification:

CLASS : PG_STUDENT.
ATTR : name < *STRING* >, roll_no < *NUMBER* >,
leave_status <{casual_leave< *NUMBER* >,
sick_leave< *NUMBER* >,
vacation_leave < *NUMBER* >} >.

Which includes the use of *structures*, the generated code would be:

```
create table PG_STUDENT (
        PG_STUDENT_id   NUMBER(p)
        name            VARCHAR2(n)
```

```
        roll_no                 NUMBER(p)
        table_name              VARCHAR2(n)
                                /* Keep table name as "leave_status" */
        create table leave_status (
                PG_STUDENT_id   NUMBER(p)
                casual_leave    NUMBER(p)
                sick_leave      NUMBER(p)
                vacation_leave  NUMBER(p)
        );
);
```

Even though the structures in the attributes are mapped to tables, they will have only one row for a particular id of the parent table, ie they will have only set of values. This can clearly be observed in the above example. Here a particular instance of the object will have only one value for the attributes in the structure. A *PG student* will have only a particular value of *casual leave,sick leave and vacation leave.*

Example:

For the specification:

CLASS : PERFORMANCE_RECORD.

ATTR : semester < *NUMBER* >, year< *DATE* >
cpi< *number* >, spi< *number* >, text< *STRING* >,
grades < TABLE[c_no < *NUMBER* >, *units* < *NUMBER* >,
grade< *CHARACTER* >] >.

This example includes *TABLE*, the generated code would be

```
create table PERFORMANCE_RECORD (
        PERFORMANCE_RECORD_id   NUMBER
        semester                NUMBER
        year                    DATE
        cpi                     NUMBER
        spi                     NUMBER
        text                    LONG(n)
```

```
        table_name                          VAR_CHAR(n)
                                            /* Keep table_name as "grades" */
        create table grades (
              PERFORMANCE_RECORD_id   NUMBER
              c_no                    NUMBER
              units                   NUMBER
              grade                   CHAR
        ):
    );
```

This type of database design can conveniently be used to access data. For example, if the grade of a particular course of a given *PERFORMANCE_RECORD_ID* is to be accessed. this can be done by the following way.

With the *PERFORMANCE_RECORD_ID*, go to the *grades* table and get the grade of the known *C_NO*.

Example:

For the specification:

CLASS : TIME_TABLE.

ATTR : time_table < TABLE [c_no < *NUMBER* >,
Instructor < *STRING* >,
schedule < TABLE [day < *STRING* >,
time_slot < [*TIME*] >] >

Where *list* is being used, the generated code would be:

```
create table TIME_TABLE (
      TIME_TABLE_id   number
      table_name      VARCHAR(n)
                      /* Keep table_name as "time_table" */
      create table time_table (
            time_table_id     NUMBER
            TIME_TABLE_id     NUMBER(p)
            c_no              NUMBER(p)
            Instructor        VARCHAR(n)
```

```
table_name        VARCHAR(n)
                  /* Keep table_name as "schedule" */
create table schedule (
        schedule_id     NUMBER
        time_table_id   NUMBER
        day             VARCHAR(n)
        table_name      VARCHAR(n)
                        /* Keep table_name as "time_slot" */
        create table time_slot (
                time_slot_id    NUMBER
                time_slot       TIME
                schedule_id     NUMBER
        );
    );
  );
);
```

Since in a *RDBMS*, it is not possible to keep structures as columns, they are mapped into tables. It is easy to retrieve information from this type of mapping, though it is time consuming. For example, if you want to find the time slot of a particular course in a particular day and if you know the *DEPARTMENT* in which the course is being offered, you can do the following steps to get the required information.

• Step 1: Get the *TIME_TABLE_id* from the table, created from the "related with" association with *DEPARTMENT*, which is dealt with later.
The corresponding PL/SQL statement would be, if the table name is *Has*

```
SELECT TIME_TABLE_id INTO time_table_id
FROM Has
WHERE DEPARTMENT_id = dept_id. /* dept_id is assumed to be known */
```

•Step 2: Go to the *TIME_TABLE* table, from there you can go to *time_table* table with values of *TIME_TABLE_id* and course number, ie *c_no*, take the *time_table_id*.

The corresponding PL/SQL statement would be:

```
SELECT table_name INTO tab_nam
                        /* table_name would be "time_table" */
FROM TIME_TABLE
WHERE TIME_TABLE_id = time_table_id
```

•Step 3: Then navigate to *schedule* table, which is given in the column *table_name.* With the *time_table_id* and the *day* column, get the *schedule_id.*

The corresponding PL/SQL statement would be

```
SELECT table_name INTO tab_nam1
                        /* table_name would be "schedule" */
FROM tab_nam /*tab_name would be "time_table" */
WHERE c_no = course_no. /* course_name is assumed to be known */
```

•Step 4: From this table, go to *time_slot* table, which is again given in the column *table_name.* With the value *schedule_id* get the related time, from the column *time_slot.*

The corresponding PL/SQL statement would be

```
SELECT table_name INTO tab_nam2
                        /* table_name would be "time_slot" */
FROM tab_nam1 /* tab_nam1 would be schedule */
WHERE day = Day /* Day is assumed to be known */
```

The use of *object ids* of other objects may be understood here.

Even though both *structures* and *tables* or *lists* are mapped to tables, the basic difference between *structure* and *tables or lists* is that, the tables formed by *structures* have only one row whereas *tables or lists* may have multiple rows for the same *object_id.*

4.1.2 RELATED WITH

This construct is also mapped into a table irrespective of the multiplicity. The possible multiplicities are:

1-1(one - one), 1-N(one - many), N-1(many - one), N-N(many - many) .

According to Rumbaugh, "an association may or may not map into a table. It depends on the database designer's preferences in terms of extensibility, number of tables and performance trade-offs" [R+90].

Any way, many to many mappings will map to a different table. This means, if there is a many to many relation between to classes, a total of three tables would be created between these two tables. One each for the two classes and one due to the relation between them.

One to many, many to one and one to one relations may or may not be mapped to a table. It may instead be merged into a class. In other words, if it is a many to one or one to many relation, table could be avoided by merging the id of the class which is having "one" side relation into the other class. In case of one to one mapping, object ids of either of the classes can be merged into the other class.

Example:

For the specification:

CLASS : STUDENT.
RELATED WITH : DEPARTMENT (Belongs_to, 1-N).

This example shows a case of one to many multiplicity.

One may generate code like this:

```
create table DEPARTMENT (
        DEPARTMENT_id    NUMBER(p)
        STUDENT_id       NUMBER(p)
);
```

In this case you can notice that STUDENT_id has been merged into DEPARTMENT's table. So, only two tables would be created, one each for STUDENT and DEPARTMENT classes, but DEPARTMENT table having the id of the STUDENT class. This way one can avoid one table by merging one classes information in the other.

But this work maps all the associations, independent of the association, into tables for the following reasons :

- Generally, associations will be between completely different objects, so it does not seem appropriate to keep a object's information in a different and independent object.
- It may not be possible to get the right multiplicity in the first few passes of the design.
- If we merge association in one class, search and update operations may become complicated.

Example:

For the specification:

```
CLASS   :   TIME_TABLE.
RELATED WITH   :   DEPARTMENT(Belongs_to, 1-1).
```

The generated code would be:

```
create table Belongs_to (
        DEPARTMENT_id    NUMBER(p)
        TIME_TABLE_id    NUMBER(p)
);
```

The advantage of using a table, for the association can be seen here. Actually this table helps in finding the exact receiving object, in sending messages. In this case if an object of DEPARTMENT is sending a message to TIME_TABLE, then to exactly which object of the TIME_TABLE, can be found out with this table. Since the table,"Belongs_to" has the ids of DEPARTMENT and TIME_TABLE, for a given id of DEPARTMENT, one can find the corresponding id of TIME_TABLE class. So, one can find the receiving object's id.

4.1.3 GENERALISATION OF and INHERITS

There are three different ways of mapping generalistation and inheritance to tables.

The normal approach is to map both superclass and sub classes to separate tables. Though this involves many tables and navigation from superclass to subclass is slow, this is a more relevant way of mapping. This is logically very clear and reasonable approach to the mapping.

The other approaches are eliminating one of super or sub classes.

In one approach the superclass is eliminated and, replicates all the attributes of the super class in all the subclasses. Though this approach observes third normal form it has some disadvantages like inability to ensure uniqueness of the attributes in the superclass. For example, if PERSON is generalising STUDENT and FACULTY_MEMBER, then in this approach you will have STUDENT and FACULTY_MEMBER having same PERSON_id which is not convenient to understand the problem. The disadvantage of this approach is illustrated in the following example.

Example:

For the specification

CLASS : PERSON.

GENERALISATION OF : STUDENT, FACULTY_MEMBER.

If the super class is eliminated, the tables generated code would be:

createtable STUDENT(

```
    PERSON_id       NUMBER(p)
);

create table FACULTY_MEMBER(
    PERSON_id       NUMBER(p)
);
```

this case, for both the tables, PERSON and FACULTY_MEMBER, PERSON_id uld start from 1, say. Then for a PERSON_id, say n, there would be two ERSONS, one a STUDENT and the other FACULTY_MEMBER.

In the third possible approach, there is only one table and all the subclasses are minated. In this approach some columns will be null for all rows. This violates ird normal form.

Of the possible three approaches this work follows the first approach given above r the reasons mentioned.

xample:

For the specification:

CLASS : PERSON.
GENERALISATION OF : STUDENT, FACULTY_MEMBER.
ATTR : name < *STRING* >, email_addr < *STRING* >, address < *STRING* >.

CLASS : STUDENT.
INHERITS : PERSON.
ATTR : roll_no < *NUMBER* >

CLASS : FACULTY_MEMBER.
INHERITS : PERSON.
ATTR : pf_no < *NUMBER* >

The generated code would be:

```
create table PERSON (
```

```
        PERSON_id       NUMBER(p)
        PERSON_type     VARCHAR2(n)
        name            VARCHAR2(n)
        email_addr      VARCHAR2(n)
        address         VARCHAR2(n)
);
create table STUDENT (
        STUDENT_id      NUMBER(p)
        PERSON_id       NUMBER(p)
        roll_no         NUMBER(p)
);
create table FACULTY_MEMBER (
        FACULTY_MEMBER_id NUMBER(p)
        PERSON_id         NUMBER(p)
        pf_no             NUMBER(p)
);
```

The navigation is done using the following steps.

- Step 1: The user gives the person's name.
- Step 2. Using this name search in the "PERSON" table get the "PERSON_id" and "PERSON_type" from the row corresponding to "name".

The PL/SQL statement would be:

```
SELECT PERSON_id, PERSON_type INTO per_id, per_type
FROM PERSON
WHERE name = 'name'.
```

- step 3. Go to the sub table with the name PERSON_type. With the "PERSON_id" get the corresponding row. The PL/SQL statement could be :

SELECT pf_no/roll_no
FROM per_type
WHERE PERSON_id = per_id.

4.1.4 AGGREGATION OF and PART OF

Aggregation is also mapped into a table. The ids of the classes, to be aggregated are stored in the table. So each object of the class to be aggregated will become the rows of the table.

Example:

For the specification:

CLASS : NOTICE_BOARD.
AGGREGATION OF : NOTICE(N).

The generated code would be

```
create table NOTICE_BOARD (
        NOTICE_BOARD_id   NUMBER(p)
        NOTICE_id         NUMBER(p)
);
```

4.2 Messages

After having mapped the objects ie static part of the problem, we will map the messages to which specify the dynamic aspect of the application.

Messages are being mapped to *procedures* or *functions*. In *ORACLE* these *procedures* and *functions* are defined using *PL/SQL* language. The messages involve *FROM* and *TO* objects, and these are mapped to tables. So, the meaning of the message is, generally to modify the *TO* table, based on the values in the *FROM* table. In this work two files with file names that of *FROM* and *TO* are created, if the file does not exist. The invocation of the *function/procedure* is done in the *FROM* file and definition of the corresponding *function/procedure* is done in the *TO*

file. In the actual implementation, calling of the message is simply invocation of a *function/procedure* and meaning of the message is defining *function/procedure* as in any of the structured languages. The definition in *RDBMS* will be just accessing the available database. Accessing includes inserting, deleting and updating the database.

In the definition of the *procedure/function*, the fact that this message has been invoked, is stored in a global table. The name of the table is "Black_board". This table shows *FROM, TO, msg_id and time_stamp.* This helps in maintaing the control flow to some extent. This would be necessary in cases where there is no way to note that a message has been sent. If every message is written on the "Black_ board", then in cases such as the above, where there is no way to find its invocation, it would help in knowing that a message has been sent. Thus, this helps in maintaing the control flow.

4.2.1 msg_id

Each *message id* is being mapped to a *function/procedure.* The parameters in the message are mapped to parameters of the *function/procedure.* In the process of mapping, an attempt is made to map parameter types to those data types which are valid in *ORACLE.* In each message, the id, which can be considered as the primary key of the sender, is included as a parameter. This would help the receiver in knowing from which object of the *FROM* class is the message sent. This will also be useful in replying.

Example:

For the specification:

FROM : DOAA.
TO : HALL_OFFICE.
msg_id : check_student_for_dues (roll_no $< NUMBER >$).
result : paid $< BOOLEAN >$

The generated code would be:

check_student_for_dues(DOAA_id NUMBER, roll_no NUMBER)

in the FROM file

and

```
CREATE FUNCTION  check_student_for_dues  (DOAA_id NUMBER, roll_no NUMBE
RETURN BOOLEAN IS
    [Local declarations]
BEGIN
 insert into Black_board VALUES (DOAA,HALL_OFFICE,check_student_for_dues, time)
 insert into HALL_OFFICE_scratch VALUES (DOAA_id)
 /* Insert code here */
END
```

in the TO file.

Mapping of *result* is explained below.

The inserting of relevant things into the table Black_board will not be shown henceforth, but it will be there in the code generated.

Parameters

Messages may have parameters. Since in *ORACLE* there are very few data types available, all the parameter types have to be mapped to the ones available. In the specification developed, you may have *tables* and *lists* in the parameters. Since there is no corresponding data structure in *ORACLE*, these types are mapped into *views*, from the classes defined in the specification. As classes, are mapped into tables, and mostly the elements in the table are the attributes of some class, which are the columns of the table formed by the class, *views* can be created. It is quite possible that more than one class may have attributes with same names. In such cases, user is given the choice of choosing from the possible options. This type of ambiguity may occur because from the specification, it is not possible to decide from which classes the elements of the table are referred. If none of the classes, contain a element of table, creation of *view* is not possible, then the fact is reported.

If a class is to be passed as a parameter, then it is reported as a table name, "Table_name" for parameter type.

Example

For the specification of message_id:

msg_id : registered_students(reg_stds $< TABLE[$
roll_no$< NUMBER >, name < STRING >] >$

The generated code would be:

The possible FROM clauses for:

Message: :registered_students: in table: reg_stds: for parm
:roll_no: are

0. STUDENT
1. REGISTRATION_FORM
2. ADD_DROP_FORM

Choose the class(Give no.):

If '0' is choosen, the code for view would be

The possible FROM clauses for:

Message: :registered_students: in table: reg_stds: for parm
:name: are

0. STUDENT
1. REGISTRATION_FORM
2. ADD_DROP_FORM

Choose the class(Give no.):

If '1' is choosen, the code for view would be

CREATE VIEW reg_stds
AS SELECT roll_no name
FROM STUDENT REGISTRATION_FORM
WHERE $<$ *constraint* $>$

Generally constraint would be null.

4.2.2 result

result decides whether a message is to be mapped to a *function* or to a *procedure*. Since a *function* in *ORACLE* returns only one value, only if a *message* results in one value, will a message be mapped to a *function*. If more than one value is to be modified, *procedures* are used with some additional work while passing the parameters. There are three different ways of passing parameters to get more than one variable modified. Which method to follow is application dependent. In this work, **IN OUT** way is used to pass the parameters in these conditions. Passing a parameter in **IN OUT** manner means that the value of the parameter being passed will be copied to the formal parameter and at the end of the *procedure*, the final value of the formal parameter is copied to the actual parameter. This is used as it can be used to get the formal parameter's value modified or just for reference. The other types of parameter passing mechanisms involve copying only in one direction.

The mapping of having a single *result* value is shown in the above (while handling msg_id) example.

Example:

For the specification:

FROM : DOAA.
TO : ACCOUNT_SECTION.
msg_id : check_student_for_dues (roll_no < *NUMBER* >).
result : payed_instt_dues < *BOOLEAN* >.

The generated code would be

check_student_for_dues (DOAA_id NUMBER, roll_no NUMBER)

in the DOAA file.

And

```
CREATE FUNCTION check_student_for_dues (DOAA_id NUMBER,
                                        roll_no NUMBER)
RETURN BOOLEAN IS
        [Local declarations]
BEGIN
```

```
insert into ACCOUNT_SECTION_scratch VALUES (DOAA_id)
/* Insert code here */
END
```

Related code may be written with in *BEGIN* and *END*.

This is for a *message* which results in a single value.

Example with a message which has no result :

Example:

For the specification:

```
FROM    :   DOAA_INTERFACE.
TO      :   DOAA.
msg_id  :   send_reg_notice_to_depts ().
```

The generated code would be:

```
send_reg_notice_to_depts ( DOAA_INTERFACE_id NUMBER)
```

in the DOOA_INTERFACE file,

and

```
CREATE PROCEDURE  send_reg_notice_to_depts (DOAA_INTERFACE_id NUMB
IS
    [Local declarations]
BEGIN
 /* Insert code here */
END
```

in DOAA file.

An example which results in more than one value :

```
FROM    :   COURSE.
TO      :   SELF.
msg_id  :   give_details (query < STRING >).
result  :   stud_detail < {name < NAME >, roll_no < NUMBER >} >.
```

The generated code would be :

give_details (give_details (COURSE_id NUMBER, query CHAR ,
name IN OUT NAME,
roll_no IN OUT NUMBER)

in COURSE file
and

```
CREATE PROCEDURE give_details (COURSE_id NUMBER, query CHAR ,
                               name IN OUT NAME,
                               roll_no IN OUT NUMBER)
IS
    [Local declarations]
BEGIN
 /* Insert code here */
END
```

in the same file, COURSE.(See notation for SELF).

Results may be in the form of *tables* and *lists*. So, they are also have to be mapped. As with parameters, the *tables* and *lists* are mapped to *views*. In case of ambiguity in dealing with *FROM* clause, choice is given to the user.

In many cases it so happens that the *result* of a message will be the parameters of the next message. This tool simply creates two similar views. So, the user has to ignore one of the views. The reason why one of *result* or *parameters* is not left is, in cases where message is being sent to itself (*SELF*), only the message which shows the *result* part is shown, and the next message, which gives results as parameters is ignored. Refer to the example specification [Sar96] for better understanding.

4.2.3 PRE_COND : Precondition

Precondition is used to get the control flow of the process at hand. This is used in the same way as *if_else* construct, which is dealt with later in the same chapter. It means only if the condition given as *PRE_COND* is satisfied, can the *FROM* object send the message, in . Actually, this is not required when the actual code is developed from the specification. *PRE & POST* (handled next) conditions are for

testing, and they would be removed from the final working system. Testing can be done by directly interpreting the specification. Testing is not within the scope of this work.

There are different types of conditions that are valid as *preconditions*. These conditions are given in the Notations chapter.

Example:

For the specification:

FROM : CDGC.
TO : STUDENT.
msg_id : replyto_give_add_drop_form (remarks < *STRING* >).
PRE_COND : done sub process validate_stud_add_drop_request.

the generated code would be:

IF DON_SUB validate_stud_add_drop_request THEN
replyto_give_add_drop_form (CDGC_id NUMBER, remarks CHAR)

Since this will be removed subsequently, we have not emphasised.

4.2.4 POST_COND : Post condition

This is similar to pre condition, but this is on the receiver side. So before receiving the message, this condition is checked on the receiver side. Only if the condition is satisfied is the message received, other wise it is ignored.

4.2.5 msg_cond: Message condition

This is treated in the same way as "pre condition".

Example:

For the specification:

FROM : CDGC.
TO : STUDENT.
msg_id : give_details (query_str < *STRING* >).

msg_cond : (date_ok == TRUE).

The generated code would be

IF (date_ok = TRUE)

give_details (CDGC_id NUMBER, query_str CHAR)

on the sender file. On the receiver side it would be dealt similarly as in earlier examples.

4.2.6 msg_type

msg_type tells, the type of the message. The possible types are: *reply, shared and exclusive*. This is valid for only those messages where msg_type is explicitly specified. This takes care most of the security. For meanings of these types, refer to the **Notation** chapter. No mapping is required for the *reply* type of messages. *Shared & exclusive* type of messages are mapped to *packages*. In *ORACLE* packages provide the required security. Special permissions to call the functions in the packages can be given. Thus security can provided. The security level of *shared* and *exclusive* types is considered to be more, they are mapped to different *packages*. For this sake, separate files, ending with "_SHR_PACKAGE" affixed to the "TO" of the message, for *shared* type of messages and "_EXEC_PACKAGE" for *exclusive* type of messages have been created to deal with the definitions of the *procedures* or *functions*. For example, if *TO* is DOAA, the file created would be DOAA_SHR_PACKAGE. for *shared* type of message and DOAA_EXEC_PACKAGE for *exclusive* type of messages. In *ORACLE packages* will provide the required security.

Except that separate files are created, no special treatment is need to be given, so example is not given.

4.2.7 Action

Action is the one which specifies what the receiver should do after receiving the message.

In the case of *sub process*, no extra processing. Simply control is transferred to the *subprocess*.

If a set of messages will clearly do a well defined function then *sub process* is introduced.

Example:

For the specification:

```
FROM    :  A.
TO      :  B.
msg_id  :  Hello ( ).
Action  :  do sub process reply.

FROM    :  A.
TO      :  B.
msg_id  :  Fine_thank_you ().
```

Sub process :: reply.

```
FROM    :  B.
TO      :  A.
msg_id  :  Hi().

FROM    :  B.
TO      :  A.
msg_id  :  How_are_you().
```

The generated code would be:

```
Hello(A_id NUMBER)
CREATE PROCEDURE  Hi (B_id NUMBER)
IS
```

```
    [Local declarations]
BEGIN
 insert into A_scratch VALUES(B_id)
 /* Insert code here */
END
CREATE PROCEDURE  How_are_you (B_id NUMBER)
IS
    [Local declarations]
BEGIN
 insert into A_scratch VALUES(B_id)
 /* Insert code here */
END
Fine_thank_you(A_id NUMBER)
```

In the file A.

```
CREATE PROCEDURE  Hello(A_id NUMBER)
IS
    [Local declarations]
BEGIN
 insert into B_scratch VALUES(A_id)
 /* Insert code here */
END
Hi(B_id NUMBER)
How_are_you (B_id NUMBER)
CREATE PROCEDURE  Fine_thank_you(A_id NUMBER)
IS
    [Local declarations]
BEGIN
 insert into B_scratch VALUES(A_id)
 /* Insert code here */
END
```

n the file B.

n case of a *generic message*, some parameters, generally, *TO* object will be pro-
ided in the *action* statement. It may also provide some constraint, generally, *TO*
onstraint, so that only those instance(s) which satisfy the constraint(s) can react
o the message. The motivation for this type of message is, if a message is to
e sent to more than one object, instead of replicating, only receiver's address is
hanged. This reduces the size of the specification and it looks more natural. In
his case also, control is transferred from the current position to the message where
he generic message is defined. After dealing with the *generic message*, control is
estored.

Example:

For the specification:

FROM : DOAA_INTERFACE.
TO : DOAA.
msg_id : send_reg_notice_to_depts ().
Action : send generic message reg_notice
{TO : DPGC, TO_constr : (ALL) },
send generic message reg_notice
{TO : DUGC, TO_constr : (ALL) }.

Generic message ::

FROM : DOAA.
TO : XXX.
msg_id : reg_notice (notice_str < *STRING* >).

The generated code would be:

send_reg_notice_to_depts (DOOA_INTERFACE_id NUMBER)

in DOAA_INTERFACE file

CREATE PROCEDURE send_reg_notice_to_depts (DOAA_INTERFACE_id NUMBER
IS
[Local declarations]

```
        BEGIN
         /* Insert code here */
        END
        reg_notice (DOAA_id NUMBER, notice_str CHAR )
        reg_notice  (DOAA_id NUMBER, notice_str CHAR )
    in the DOAA file.
    /* ONLY BY ALL */
         CREATE PROCEDURE   reg_notice (DOAA_id NUMBER, notice_str CHAR )
         IS
            [Local declarations]
         BEGIN
          insert into DPGC_scratch VALUES (DOAA_id)
          /* Insert code here */
         END
     in DPGC file
and
     (ONLY BY ALL)
         CREATE PROCEDURE   reg_notice (DOAA_id NUMBER, notice_str CHAR )
         IS
            [Local declarations]
         BEGIN
          insert into DUGC_scratch VALUES (DOAA_id)
          /* Insert code here */
         END

 in DUGC file.
```

Here the message "reg_notice" is sent to all *DPGC*s and *DUGC*s with one message.

In case of a *generic sub process*, a set of *messages* have to be processed, instead of one *message* as happens in the case of a *generic message*. Since this is a generic sub process, it may be invoked in different circumstances. So, some parameters are

to be provided, in order to make it flexible. So, at the definition of the *generic sub process*, a parameter similar to that of formal parameter, is declared. It is declared with keyword **INPUT**. The corresponding actual parameter is passed in the *action*. So, all the instances of the formal parameter are replaced by the actual parameter. The rest is same as in the above cases.

Example:

For the specification:

FROM : STUDENT.
TO : DPGC.
msg_id : sign_reg_form (reg_form < *REGISTRATION_FORM* >).
result : reg_form < *REGISTRATION_FORM* >.
Action : do generic sub process check_and_sign_reg_form (reg_form < *REGISTRATION_FORM* >) {CONVENER:DPGC}.

FROM : STUDENT.
TO : DUGC.
msg_id : sign_reg_form (reg_form < *REGESTRATION_FORM* >).
result : reg_form < *REGESTRATION_FORM* >.
Action : do generic sub process check_and_sign_reg_form (reg_form < *REGESTRATION_FORM* >) {CONVENER:DUGC}.

Generic sub process :: check_and_sign_reg_form.
Input : CONVENER.
FROM : CONVENER.
TO : SELF.
msg_id : check_reg_form (reg_form < *REGISTRATION_FORM* >).

FROM : CONVENER.
TO : REG_FORM.
msg_id : put_convener_sign (signature < *SIGNATURE* >).
msg_type : safe.

The generated code would be:

reg_form := sign_reg_form(STUDENT_id NUMBER,
reg_form :Table_name:)
reg_form := sign_reg_form(STUDENT_id NUMBER,
reg_form :Table_name:)

in the STUDENT file,

```
CREATE FUNCTION  sign_reg_form (STUDENT_id NUMBER,
                                reg_form Table_name )
RETURN REGISTRATION_FORM IS
        [Local declarations]
BEGIN
 insert into DPGC_scratch VALUES (STUDENT_id )
 /* Insert code here */
END

check_reg_form (DPGC_id NUMBER, reg_form Table_name )
CREATE FUNCTION  check_reg_form (DPGC_id NUMBER,
                                 reg_form Table_name )
RETURN BOOLEAN IS
        [Local declarations]
BEGIN
 /* Insert code here */
END
put_convener_sign (DPGC_id NUMBER, signature SIGNATURE )
```

in the DPGC file,

```
CREATE FUNCTION  sign_reg_form (STUDENT_id NUMBER,
                                reg_form Table_name )
RETURN REGISTRATION_FORM IS
        [Local declarations]
BEGIN
```

insert into DUGC_scratch VALUES (STUDENT_id)
/* Insert code here */
END
check_reg_form (DUGC_id NUMBER, reg_form Table_name)

CREATE FUNCTION check_reg_form (DUGC_id NUMBER,
reg_form REGISTRATION_FORM)
RETURNBOOLEAN IS
[Local declarations]
BEGIN
/* Insert code here */
END
put_convener_sign (DUGC_id NUMBER,signature SIGNATURE)

in the DUGC file and

CREATE PROCEDURE put_convener_sign (REG_FORM_id NUMBER,
signature SIGNATURE)
IS
[Local declarations]
BEGIN
/* Insert code here */
END
CREATE PROCEDURE put_convener_sign (REG_FORM_id NUMBER,
signature SIGNATURE)
IS
[Local declarations]
BEGIN
/* Insert code here */
END

in the REG_FORM file.

4.2.8 TO CONSTRAINTS

There are different ways of specifying the *TO* constraints. They are discussed below:

• variable == class::attribute.

The above constraint means that only those objects which satisfy the constraint, are to be affected due to the above message. This is mapped to the *WHERE* clause of the SQL query. But here we do not include it as part of the code. Instead this is indicated through a comment in the definition of the *procedure/function.*

Example:

For the specification:

```
FROM    : hspacelem DOAA.
TO      : hspacelem COURSE (c_no == COURSE::c_no).
msg_id  : hspacelem give_handle().
```

The generated code would be

```
give_handle (DOAA_id NUMBER)
```

in the DOOA file

and

```
/* only by c_no:=:COURSE:c_no: */
CREATE PROCEDURE give_handle (DOOA_id NUMBER)
IS
  [Local declarations]
BEGIN
 insert into COURSE_scratch VALUES (DOAA_id)
 /* Insert code here */
END
```

in the COURSE file.

• class::attribute1 <=> class::attribute2.

This is handled in the same the way as the earlier case.

• ALL

This constraint means that the message in consideration is relevant to all the objects

of the *TO* class. In the translation *WHERE* clause can be omitted. In the current translation just a comment is produced.

- ALL_RELATED.

This construct means that the current message is relevant to all the objects of the *TO* class satisfying some condition. For example, a COURSE may send a message to STUDENT for some information. Then the *TO* constraint could be ALL_RELATED. This means that all instances of the STUDENT class who have registered for the COURSE should respond to the message. The constraint can be written from the "related with" table of the STUDENT and COURSE classes.
The comment would be /* ALL_RELATED */

- handle == BACK.

Before this constraint occurs, some object of the *TO* class would have sent a message, to the *FROM* class, for which the object would be expecting a reply. So, this constraint means that the current message is meant for the object which sent the previous message.

The comment would be /* TO_id in consideration */. In the mapped output the value of the TO class would be displayed.
In the actual implementation, the *FROM* of the message in consideration, is supposed to have stored the object id of the *TO* class, in some scratch space in its table, when this *TO* had sent a message to this *FROM* class. So, this *FROM* could sent to the actual destination.

Example:

For the specification:

FROM : STUDENT.
TO : DOAA.
msg_id : req_for_reg(reg_form ¡REG_FORM¿.
result : remarks.

FROM : DOAA.
TO : STUDENT(handle == BACK).

msg_id : replyto_req_for_reg_form(remarks ¡STRING¿).

The generated code would be:

```
remarks = req_for_reg(STUDENT_id NUMBER, reg_form REG_FORM).
CREATE PROCEDURE replyto_req_for_reg_form(DOAA_id NUMBER, remarks CHAR
IS
   [Local declarations]
BEGIN
 insert into STUDENT_scratch VALUES (DOAA_id)
 /* Insert code here */
END
```

in the STUDENT file.

```
CREATE FUNCTION req_for_reg(STUDENT_id NUMBER, reg_form Table_name)
RETURN REG_FORM IS
     [Local declarations]
BEGIN
 insert into DOAA_scratch VALUES (STUDENT_id)
 /* Insert code here */
END
```

replyto_req_for_reg_form(DOAA_id NUMBER, remarks CHAR)

in the DOAA file.

4.2.9 if else

if statement has the same semantics as in languages like *C, Pascal etc.* It means if the condition in the *if* is **TRUE** process the messages given below until you encounter end of the *if* condition. *if* may be followed by *else* which means if the above condition evaluates to **FALSE** process the messages below *else* until end of the *else* condition is reached. Syntax is given in the **Notation** chapter.

Example:

For the specification:

```
    FROM    :   DOAA.
    TO      :   ACCOUNT_SECTION.
    msg_id  :   check_student_for_dues (roll_no < NUMBER >).
    result  :   payed_instt_dues < BOOLEAN >.
if (payed_instt_dues   ==   TRUE) {
    FROM    :   DOAA.
    TO      :   SELF.
    msg_id  :   form_links ().
} else {
    FROM    :   DOAA.
    TO      :   STUDENT.
    msg_id  :   replyto_req_for_reg (Instt_dues_not_paid < STRING >).
}
```

The generated code would be:

```
    payed_instt_dues   =:    check_student_for_dues (DOAA_id NUMBER, roll_no NUMBE
IF ( payed_instt_dues   =   TRUE )
    form_links (DOAA_id NUMBER)
    CREATE PROCEDURE  form_links (DOAA_id NUMBER)
    IS
        [Local declarations]
    BEGIN
     /* Insert code here */
    END
ELSE
    replyto_req_for_reg (Instt_dues_not_paid CHAR )
```

in the DOAA file ,

```
    CREATE FUNCTION  check_student_for_dues (DOAA_id NUMBER, roll_no NUMBE
    RETURN BOOLEAN IS
```

[Local declarations]
BEGIN
insert into ACCOUNT_SECTION_scratch VALUES (DOAA_id)
/* Insert code here */
END
in the ACCOUNT_SECTION file and
CREATE PROCEDURE replyto_req_for_reg (DOAA_id NUMBER,
Instt_dues_not_pa
IS
[Local declaratio
BEGIN
insert into STUDENT_scratch VALUES (DOOA_id)
/* Insert code here */
END
in the STUDENT file.

4.2.10 foreach

Till now we have been dealing with cases where each *message* is to be executed only once. In some cases it may be required to give same type of treatment to a set of objects. In such cases a set of *messages* have to be sent/received to/from a set of objects. For each *message*, *FROM* and *TO* objects will be different. These objects are choosen from a *list*. Since this *list* will be a result or parameter of some previous *message* and as they have already been mapped into *views*, the *for loop* can pick objects from these *views*.

If for a set of students we want the courses they are doing (assuming that all these studentsare in a list textitstudent), the specification would be

Example:

```
foreach stdnt (student)
{
        FROM   :   DPGC.
```

TO : STUDENT(stdnt == STUDENT::stdnt).
msg_id : give_course_no().
result : courses< [*c_no*] >.

FROM : STUDENT(handle == BACK).
TO : DPGC.
msg_id : replyto_give_courses(courses< [*c_no*] >).

}

The generated code would be

SELECT COUNT(*) INTO x
FROM student
FORi in 1 to x
SELECT stdnt INTO stdnt_temp
FROM students
WHERE ROWNUM = i
< *The code similar to the one for messages is to be inserted* here >.

Chapter 5

Conclusion, Limitations And Related Work

In this thesis we have described a translator for a message centric OO specification language to a RDBMS. We also indicate what other relevant work is on and point the limitations.

5.1 Limitations

In this work, we tried to map as many constructs as possible. Some types of attributes like "signature" could not be mapped. This could not be done as there is no corresponding data type in *ORACLE*. In the actual implementation, this type may not be necessary.

In testing the specification, while testing the validity of the parameters, this work tests only with in the same *process, sub process* or *generic sub process*, but to confirm properly, we need to check a little more. This can be done, but testing of the specification is not emphasised.

While checking the validity of the parameters, even the index of the loops are to be considered which is not taken care of.

Completely different ways of checking the validity of the parameters used by the "Interface" objects are to be followed. This work does not even parse the "Interface"

objects. So, testing of the interface objects is ignored.

Sorting is done based on the actions of the messages. Messages not having actions are ignored. So the message following immediately, in the specification, such messages, which doesn't have actions, will remain as it is.

Also, it is assumed that the message which occurs first in the process of specification is the starting message of the process.

5.2 Related work

Parallel to this work of mapping the specification into *ORACLE* templates which is a non object oriented language, mapping of the same to *C++* which is a object oriented language, is being done by D. Rafee. Though this work does some testing, main testing is proposed to be done later. The specification has been developed by Sarma.

5.3 Conclusions

This work mainly designs the database and generates code to generate tables for object the model and procedural templates for the message model of the specification. Though it does not provide the actual code, these templates will be helpful in writing the code. We also tried to report some errors in the specification. This work helps in extending the work such as automatic testing.

All the mappings are shown in a table, next page, for quick reference.

Constructs in Specification	*Mapping in Oracle*
CLASS	Table
ATTR	columns
structure in the attrs.	Table
TABLE in the attrs.	Table
list in the attrs.	Table
Generalisation & Inheritance	Table
RELATED WITH	Table
AGGREGATION & PART OF	Table
FROM	File
TO	File
msg_id	Proc/Func name
Result with one value	Function
No Result	Procedeure
Result with more than one value	Procedure
Execlusive & shared in msg_type	Package
Msg_cond	IF
Pre_cond	IF
Post_cond	IF
if	IF
else	ELSE
foreach	FOR
parameter	parameter
TABLE in the parameters	view
List in the parameters	view

Appendix A

BNF NOTATION

```
spec        :  /* empty */
            |  spec clormsg_spec
            ;

/* Parsing of class specification */

clormsg_spec: CLSS ':' Name opt
              |  proclist
              ;
Name        :   TEXT '.'
            ;
opt         : /*empty*/
            |    TYPE ':' TEXT '.' opt
            |    GEN_OF ':' gen_list opt
            |    ATR ':'attr_list opt
            |    INHTS':' inhr_list opt
            |    RLTDS':'  rel_list opt
            |    AGG':'agg_list opt
            |    CRTS':' creates opt
```

```
                | rel_list ',' rel '.'
                ;
rel             : TEXT '('link mtcty ')'
                ;
link            : TEXT ','
                | TEXT link ','
                ;
mtcty           :  TEXT
                ;

/* Parsing of process */

proclist        :    proc
                |    proclist    proc
                ;
proc            :  PROC ':' ':' TEXT '.' subormsg_list
                ;

/* Parsing of messages */

subormsg_list   :  subormsg_list subormsg
                | /* empty */
                ;
subormsg        :    FRM ':' TEXT frm_const '.'
                     TO ':' TEXT to_const '.'
                     MSG':'TEXT '('parm_list option
                |    loop_cond
                |    GEN_PROC':' ':' TEXT '.' INP ':' TEXT '.'
                |    GEN_MSG ':' ':'
                |    SUB_PROC ':' ':' TEXT '.'
                ;
```

```
loop_cond       :    FOR TEXT '(' for_parm ')' '{' subormsg_list '}'
                |    if_else
                ;
if_else         :    IF '(' Bool_expr ')' '{' subormsg_list '}' if_else_cont
                ;
Bool_expr       :    Bool_expr and_or Bool_expr
                |  UMINUS  Bool_expr
                |  '(' Bool_expr ')'
                |  rel_expr
                ;
rel_expr        :  TEXT if_op Tr_Fa
                ;
Tr_Fa           :    TRU
                |    FALS
                |    TEXT
                ;
if_op           :    EQL
                |    LEQ
                |    GEQ
                |    LS
                |    GT
                |    NEQ
                ;
if_else_cont    :  /* empty */
                | ELSE '{' subormsg_list '}'
                ;
for_parm        :  lst_pln_opn TEXT lst_pln_cls
                ;
lst_pln_opn     :   '['
                |   /* Empty */
                ;
```

```
lst_pln_cls     :   ']'
                |   /*empty*/
                ;
to_const        :   '('TEXT EQL TEXT ':' ':' TEXT')'
                |   '(' TEXT ':'':' TEXT rel_op TEXT ':'':'TEXT ')'
                |   '('HND EQL BAK')'
                |   '(' all_rltd ')'
                |   /* empty */
                ;
all_rltd        :  ALL
                | ALL_RLTD
                ;
frm_const       : '('TEXT EQ TEXT')'
                |   /* empty */
                ;
rel_op          :     GT
                |     EQ
                |     LS
                ;
parm_list       :    parm ')' '.'
                |    parm_list ',' parm ')' '.'
                ;
parm            :    val_par TEXT LS  parm_type GT
                |     /*empty*/
                ;
val_par         :   '$'
                |   /* empty */
                ;
parm_type       :    '[' list_table_pln ']'
                |    list_table_pln
                ;
```

```
list_table_pln :    TABL '[' tab_parm_list
               |    TEXT
               ;
tab_parm_list :    parm ']'
               |    tab_parm_list ',' parm ']'
               ;
option         :    /*empty*/
               |    MSCND ''(' condition_list  option
               |    RESU':'   res_list  option
               |    PRE ':' pre_post_cond_list option
               |    POST ':' pre_post_cond_list option
               |    ACTS':' Action option
               |  MG_TYP ':' msg_typ '.' option
               ;
msg_typ        :  SAFE
               |  EXEC
               |  SHRD
               |  RPLY
               |  FRWRD
               ;
pre_post_cond_list : pre_post_cond '.'
                     | pre_post_cond_list and_or pre_post_cond '.'
                    ;
pre_post_cond      :  RCVD TEXT
                   |  RCVD RES_OF TEXT
                   |  DONE S_PROC TEXT
                   |  RCVD TEXT OF TEXT
                   |  RCVD RES_OF TEXT OF TEXT
                   |  TEXT if_op Tr_Fa
                   ;
Action         : action '.'
```

```
                | action ',' Action '.'
                |/* empty */
                ;
action          : DO GEN_SUB_PROC action_text action_parm  sub_proc_const
                | SND_GEN_MSG action_text action_parm  msg_const
                | sub_pln action_text action_parm sub_proc_const
                | SLCT act_text
                | RTR action_text action_parm
                | DISPLAY TEXT
                ;
sub_pln         :    SND_MSG
                |    DO S_PROC
                ;
action_text     :     TEXT
                ;
act_text        :    TEXT act_text
                |    TEXT
                ;
action_parm     :   '(' act_parm_list ')'
                |   /* empty */
                ;
act_parm_list   :    act_parm
                |    act_parm_list ',' act_parm
                ;
act_parm        :    TEXT LS TEXT  GT
                ;
sub_proc_const  :   '{' sp_const_lst '}'
                |    /* empty */
                ;
sp_const_lst    :    sp_const
                |    sp_const_lst ',' sp_const
```

```
                ;
sp_const        :    src':'Target
                ;
src             :    TEXT
                |    FRM
                |    TO
                ;
Target          :    TEXT
                |    '$'TO
                | '$'FRM
                ;
msg_const       :   '{' msg_const_list '}'
                ;
msg_const_list  :    msg_constraint
                |    msg_const_list ',' msg_constraint
                ;
msg_constraint  :    Tar ':' Src
                ;
Tar             :    TO
                |    TOC
                ;
Src             :    TEXT
                |    '(' ALL ')'
                |    '(' BAK  ')'
                ;
condition_list  :    condition ')' '.'
                |    condition_list and_or condition ')' '.'
                ;
condition       :    TEXT if_op Tr_Fa
                |    ')' '.'
                ;
```

```
and_or          :    AND
                |    OR
                ;
res_list        :    result '.'
                |    res_list ',' result '.'
                ;
result          :    TEXT LS res_typ GT
                |    /* empty */
                ;
res_typ         :    '[' list_tab ']'
                |    '{' tab_res_list
                |    list_tab
                ;
list_tab        :    TABL'['tab_ res_list
                |    TEXT
                ;
tab_res_list    :    result tab_cls
                |    tab_res_list ',' result tab_cls
                ;
tab_cls         :    '}'
                |    ']'
                ;
```

Appendix B

Errors Messages

This appendix gives the description of the error messages that are reported.

B.1 Object Specification

This section gives the errors reported for the object specification.

Error: Class :class_name: exists

The class "class_name" is defined more than once.

Example:

CLASS : PERSON.
ATTR : name< $STRING$ >, id < $NUMBER$ >.

CLASS : PERSON.
ATTR : name< $STRING$ >.

Error: Attribute :attr: exists in class :class_name:

The attribute "attr" is declared more than once in the class "class_name":

Example:

CLASS : PERSON.
ATTR : name< $STRING$ >, ..., name< $STRING$ >

Error: There is no class as :name: from which :class_name1: can inherit

The class "class_name1" is defined to be *inheriting* from "name" which is not defined in the object specification.

Example:

```
CLASS   :  STUDENT.
INHERITS   :  PERSON.
```

If PERSON is not defined, this error is reported.

Error: The class :class_name2: is not generalising :class_name1:

The class "class_name1" is defined to be *inheriting* "class_name2". But "class_name2" is not defined to be *generalising* "class_name1".

Example:

```
CLASS   :  STUDENT.
INHERITS   :  PERSON.

CLASS   :  PERSON.
GENERALISATION OF   :  FACULTY_MEMBER.
```

Error: There is no class as :name: which :class_name: can generalise

The class "class_name" is defined to be *generalising* "name" which is not defined in the object specification.

Example:

```
CLASS   :  PERSON.
GENERALISATION OF   :  FACULTY_MEMBER.
```

If FACULTY_MEMBER is not defined, this error is reported.

Error: The class :class_name2: is not inheriting :class_name1:

The class "class_name1" is defined to be *generalising* "class_name2". But "class_name2" is not defined to be *inheriting* "class_name1".

Example:

CLASS : PERSON.
GENERALISATION OF : STUDENT

CLASS : STUDENT.
ATTR : name< *STRING* >.

Error: There is no class as :name: to which :class_name: can relate
The class "class_name" is defined to be *related with* "name" which is not defined in the object specification.

Example:

CLASS : STUDENT.
RELATED WITH : PERFORMANCE_RECORD.

If PERFORMANCE_RECORD is not defined, this error is reported.

Warning: The class :class_name2: has no relation with :class_name1: The class "class_name1" is defined to be *related with* "class_name2", but "class_name2" is not defined to be *related with* "class_name1".

Example:

CLASS : STUDENT.
RELATED WITH : PERFORMANCE_RECORD.

CLASS : PERFORMANCE_RECORD.
ATTR : cpi< *NUMBER* >.

Error: There is no class :name: which :class_name: can aggregate
If the class "class_name" is defined as the *aggregation of* "name" which is not defined in the object specification, this error is reported.

Example:

CLASS : NOTICE_BOARD.
AGGREGATION OF : NOTICE.

If NOTICE is not defined, this error is reported.

Error: The class :class_name2: is not a part of :class_name1:
The class "class_name1" is defined as the *aggregation* of "class_name2" but "class_name2" is not defined as a *part of* of "class_name1".

Example:

```
CLASS   :  NOTICE_BOARD.
AGGREGATION OF   :  NOTICE.

CLASS   :  NOTICE.
```

Error: There is no class as :name: for which :class_name: can be a part of
The class "class_name" is defined to be a *part of* "name" which is not defined in the object specification.

Example:

```
CLASS   :  NOTICE.
PART OF   :  NOTICE_BOARD.
```

If NOTICE_BOARD is not defined, this error is reported.

Error: The class :class_name2: is not aggregating :class_name1:
The class "class_name1" is defined as *aggregation of* "class_name2", but "class_name2" is not defined as a *part of* "class_name1".

Example:

```
CLASS   :  NOTICE.

CLASS   :  NOTICE_BOARD.
AGGREGATION OF   :  NOTICE.
```

B.2 Message Specification

This section gives the errors reported for the message specification.

Error: There is no class :name: from which you can send a mesg

If there is message for which "name" is the *FROM*, this error is reported.

Example:

FROM : STUDENT.
TO : DOAA.
msg_id : req_for_reg_form.

If STUDENT is not defined in the object specification, this error is reported.

Error: There is no class :name: to which you can send a mesg

If there is message for which "name" is the *TO*, this error is reported.

Example:

FROM : STUDENT.
TO : DOAA.
msg_id : req_for_reg_form.

If DOAA is not defined in the object specification, this error is reported.

Error: The parameter :parm: in ::mesg:: by ::from:: is not valid

If the parameter "parm" used by "from" in the message "mesg" is not valid, this error is reported.

A parameter is said to be valid if it is from either one of the following:

1. A attribute of the FROM class.
2. Received as a result previously.
3. Received as a parameter previously.

Error: The action :Action: is :type: not defined

The action "Action" is defined to be done in a message, but the action is not defined. The type of the action is "type". The possible types are "sub",a sub process, "gen", a generic sub process, "gen_msg", a generic message and "simple", a ordinary message .

Appendix C

Notation

C.1 Object model

The object model is used to represent the static structure of the system. There are three kinds of objects in the system : *entity objects, interface objects and control objects.* entity or data objects are used to store information. They usually exist in the problem domain and represent some real-world entity or concept. Interface objects are used to view data in entity objects. Control objects are used to collaborate between several entity and interface objects. The notation for entity and control objects is same and is different from notation for interface objects.

C.1.1 Class Specification

This specification is valid for *entity* and *control* objects.

CLASS : *Name of the class.*

Name of the class is an identifier[1]. This declaration begins declaration of a new class. All the items that follow this declaration are assumed to belong to this class. Only another "CLASS : xxx" or end-of-file will end the declaration of this class.

TYPE : *ABSTRACT* .

[1]Syntactically, an identifier begins with a letter [a-zA-Z_] and contains any alphanumeric character (including "_").

this field is optional. if nothing is specified about the **TYPE**, then it will be taken as non-abstract data class.

An abstract class is one which does not have any instance. A class can be an abstract class only if it is a base class of an inheritance hierarchy.

For example, in the DOAA system, **Person** is an abstract class whose derived classes are STUDENT, FACULTY_MEMBER, DOAA etc. A class derived from an abstract class can itself be an abstract class.

INHERITS : *class name [,class name ...]*
The list of classes from which this class inherits.

ATTR : *attr_name <type> [, attr_name <type> ...]*
list of attributes each specified as *attribute_name <type>*. *type* can be any of

- *STRING, NUMBER, DATE.. etc.* (i.e. scalar type)
- *[attr_decl]* (represents list)
- *TABLE [attributes list]* (a table)
- { *attributes_list* } (set of dissimilar but related items)

Examples : Person has attribute *name.*

name <STRING>

Grade_Sheet has table of course number and grade.

grades_list < TABLE [course_no <STRING>, grade <CHARACTER>] >

GENERALIZATION OF : *class_name [, class_name ...]*
class_name [, class_name ...] are classes which inherit this class.

AGGREGATION OF :
Class name (multiplicity) [, Class name (multiplicity)].
PART OF : *Class name (multiplicity)[, Class name (multiplicity)].*

multiplicity is the number of part of objects of the given class in the aggregate object . Note that multiplicity and class names are specified for part-of objects also. This means that there can be shared aggregation.

For Example, Department is collection of Students, Faculty members, etc. But a Faculty member can belong to more than one department. So we have to specify more than one class.

Consider a geometric system where a square is modeled as composition of triangles. In this case, same triangle can be part-of more than one square. So, we need to have multiplicity in the **PART OF** specification.

CLASS : *DEPARTMENT*
AGGREGATION OF : *STUDENT* (*N*), *FACULTY* (*N*).

CLASS : *STUDENT.*
PART OF : *DEPARTMENT (1).*

CLASS : *FACULTY_MEMBER.*
PART OF : *DEPARTMENT (N).*

Here, a STUDENT belongs to only one Department but a Faculty Member can belong to more than one Department.

RELATED WITH :

Related Class name (relation_name, multiplicity)
[, *Related Class name (relation_name, multiplicity)*]

relation_name is the role the related object plays with respect to this object in the relation.

multiplicity specifies how many instances of the Related Class are related with one instance of this class.

Here, the relation is a uni-directional relation. So, a relation specification *B (xxx, y)* in Class *A* means that B is related to A. But not that *A* is related to *B*.

If A is also related to B, then we have to specify in B's Class specification. This is necessary because not all the relations are two-way relations (Ex: Relations like *uses*, *creates* etc.)

C.1.2 Interface object specification

CLASS NAME : *Name of the interface class*

TYPE : *INTERFACE*

MENU ITEMS :

{ *id : "menu_identifier"*,
name : "name of the menu",
action : Action to be taken when this menu item is selected,
description : help message,
} [,{...}]

menu_identifier is used to uniquely identify this menu item. In addition to normal identifier syntax, "/" symbol can be used to denote the hierarchy of menu items.

name is the name of the menu item in more detail. This name is displayed to the user.

description is the message to be displayed when the user asks for help about this menu item.

Action will be one of

- send message msg_id [of process_id [to object]]
- do sub_process subproc_id
- invoke_menu menu_id
- select menu_item "some_other_menu_item"
- select fillin_item "a_fillin_item"

The actions are explained in **Message specifications.**

For Example, some menu items of the **STUDENT_INTERFACE** object are shown below.

CLASS : *STUDENT_INTERFACE*

TYPE : *INTERFACE*

INHERITS : *PERSON_INTERFACE*

MENU ITEMS :

{ *id : "add_drop"*,
name : "Add / Drop Courses",
action : invoke_menu [*"add_drop/get_add_drop_form"*,
"add_drop/fill_add_drop_form",
"add_drop/submit_add_drop_form"]

},
{ id : "register/send_reg_form",
name : "Submit Registration Form",
action : send_message submit_filled_reg_form of Registration_process
description : Invoke this if you have filled in the
add/drop form and want to submit it

}

FILL_IN ITEMS :

{ *id : "fillin_identifier"*,
name : "name of fill in item",
constr : Constraints on the value of this fillin item,
action : Action to be taken when this item is filled,
attached_to : is this item attached to an attribute of some other class ?

}

constraints can used to define either type constraints or value constraints on this fill_in item.

Attached to will be useful to initialize the values of fill_in items when required.

For example, in the Registration form interface, When student selects fill registration form, blank registration form will be displayed. But when he selects modify_reg_form, the values of fill_in items are initialized to corresponding values of their attached objects. (**STUDENT** for *name*, **REG_FORM** for *courses*, etc)

fill_in items are generally used to represent data in forms and to take some temporary input data.

For example, a fill_in item in REGISTRATIONFORM_INTERFACE is given below.

{ *id : "student_name"*,
name : "Name in Block letters",
constr : type = <STRING>,
attached to : STUDENT
}.

DISPLAY METHOD : These are used to specify logical display of screen with fill_in and menu items.

INHERITS :
GENERALIZATION OF :

Semantics of inheritance for interface classes differs slightly from that for normal classes. If interface class A inherits from interface class B, then A can use or replace some or all *fillin* items and display methods of B, and can also extend those menu items of B whose "actions" are invoke_menus.

For example, in **PERSON_INTERFACE**, the *queries* menu item will have its action as

invoke_menu ["queries/see_calendar",
"queries/see_notice_board"]

STUDENT_INTERFACE inherits **PERSON_INTERFACE**, and extends this menu_item to include some more sub-menu items given below.

invoke_menu ["queries/see_offered_courses",
"queries/see_other_student_details"]

RELATED WITH : *CLASS (relation, multiplicity)*

This has the same semantics as in normal class definitions.

CLASS in RELATED WITH is used to declare which classes the interface object needs to attach to its fill_in items.

C.2 Message model

process :: *process_name.*

sub process :: *sub_process_name.*

we divide something into sub processes when there is a central object that is gathering data from several objects to do certain operations.

generic sub process :: *generic_sub_process (parameters).*

INPUTS : parameters_varying_for_generic_sub_process.

The declaration of a set of messages begins with one of the above three **process**, **sub process** or **generic sub process** declarations.

FROM : *Class name.*

TO : *Class name (constraint).*

Class name is the name of the class specified in the object model.

constraint can be

- condition on some attribute of the object. This is used to identify particular instance of the class.

  (variable == class::attribute)
  (class::attribute1 <=> class::attribute2)

- ALL, meaning all the instances of the class should be used for the purpose.

- ALL_RELATED, meaning all the instances receiver class related to this object.

- handle == ref
  handle == BACK

  The above means that we explicitly give the handle (*handle* identifies an object in the system. Every object will have an unique handle. In the context of a Programming language like $C++$, handle is same as reference. In the context of a Database, handle is the primary key) instead of fetching it from the relations of the sender object.

  When the handle is BACK, this means that the sender should not pick up the object handle from its Relations table, but should just send it, as a response to the message it received, to the corresponding object. For example, any Student can query a Course for its details. Then the Course will extract the details and sends it to Student. While replying, if we specify the Constraint as *name == xxx*, Course will just see that the Class STUDENT is related to it and checks whether this particular STUDENT exists in its Relations table. If not so, it will not send the message. So, we should specify that even though this Student is not related to the Course, it knows his address through the prior request.

msg_id : *message identifier (parameters).*

The syntax of *message identifier* is same as that of *class name* in **Class specification.** The syntax of the parameters is the same as syntax of "list of attributes" as described in the **Class specifications.**

msg_type : *type of the message.*

type of the message can be reply, shared or exclusive.

reply means the message is being sent as a reply to a message this object has previously received. The sender of the request message expects and waits for this reply message.

shared and *exclusive* represent the security level of the message. If it is specified as *exclusive*, then only this object can send the message to the receiver. If it is specified as *shared*, then other objects can send this message to the receiver only

if security level in that message specification is also specified as *shared. shared* is useful to restrict the senders to a set of objects.

msg_cond : *constraint that does not depend on the state of the sender, to send the message.*

This is used to send a message repeatedly (looping of the message) or send a message depending on some condition which is not associated with the state of the object.

Example :

- if (variable == value)
- foreach item (table_name or list_name)

For reasons of clarity, these message conditions surround the entire thread of messages which are invoked in a sequence as a result of issuing this message. (See ?? for example)

result : *reply to this message*

The syntax of "result" is similar to that of list of attributes.

When you specify the "result", it means that the sender "expects" some response to this message. This response is the result of action taken by the receiver of the message.

PRE_COND : *condition on the state of the sender before sending the message*

POST_COND : *condition on the state of the receiver before receiving the message*

It is the combination (and/or) of one or more of the following things.

- received msg_id [of process_id]
- received result_of msg_id [of process_id]
- sent msg_id [of process_id]

- attribute <=> value
- attribute1 <=> attribute2

The *Pre condition* is a constraint on the state of the sender to send the message. *Post condition* is a constraint on the state of the receiver to receive the message.

By specifying different Post conditions for the same message exchanged between same two objects, we can have different actions depending on when the message is received. Other use of Post condition is to decide whether to accept or reject the message depending on its time of arrival.

For Example, when a student sends a registration form request to DPGC, depending on whether he had received Registration forms from DOAA or not, he will process the message or ignores it.

While coding, *Pre condition* is generally useful to position the function call.

Pre and Post conditions are also important for testing purposes.

Action : *action to be taken after this message has been received*

Action can be any of the following

- send message msg_id [of process_id]
- do sub_process sub_proc_id (parameters)
- send generic message msg_id {TO:value, FROM:value, ... etc.}
- do generic sub_process process_id (parameters) {values to substitute}
- select menu_item xxx

"select menu_item xxx" can be used only if the receiver is an Interface object.

generic message ::

declaration of normal message, except that some fields will be empty.

C.2.1 Variables in Messages

To specify the source of a variable (an argument to the Message or a variable in a PRE-COND etc.), we use the following :

- an attribute of sender ⟶ $variable.
- Sender receives it as an argument of some other message of this process ⟶ $msg_id!variable .
- Sender receives it as an argument of some other message of some other process ⟶ $proc_id!msg_id!variable .
- Sender receives as a result of the message that was sent by it in this process ⟶ $proc_id!msg_id=variable .

  The following two messages from *get_student_details* sub process of *Evaluation_process* illustrate the notation.

FROM	: COURSE.
TO	: STUDENT (ALL_RELATED).
msg_id	: give_details
($Evaluation_process!give_registered_students!query <STRING>).	
result	: details <{name <NAME>, roll_no <NUMBER>}>.

The "query" variable used in the argument of this message is received by COURSE as the argument of the message give_registered_students of the process Evaluation_proces

FROM	: STUDENT.
TO	: COURSE
msg_id	: replyto_give_details ($name <NAME>, $roll_no <NUMBE
msg_type	: reply.
Action	: send message build_student_details_table.

STUDENT replies to the query with the arguments being its attributes.

```
FROM        : COURSE.
TO          : SELF.
msg_id      : build_stud_details_table
($give_details=name<NAME>, $give_details=roll_no <NUMBER>).
PRE_COND    : received result_of give_details.
result      : registered_students <TABLE [ name <NAME>,
                                           roll_no<NUMBER>]>.
```

- If it is a loop variable (Eg. foreach loop etc.) ⟶ $nest_level~variable .

```
foreach course_no (course_grades_not_received) {
        FROM        : DOAA.
        TO          : COURSE (COURSE::c_no == $1~course_no).
        msg_id      : inform_instructors_to_send_grades
                      (msg_str = "Please send grades" <STRING>).
}
```

- If the sender of the message is an Interface Object, then the interface object may use some data which it has got as temporary input. this data can be represented by ⟶ %variables_list.

- Internal values of message fields can referred by prefixing the field with "$". This can be useful in generic processes.

  Eg. $FROM, $TO, $FROM_constr, etc.

  In the following example, when the menu item "prepare_notice/registration" is selected from DOAA_INTERFACE object it will send a message to NOTICE_INTERFACE object to prepare the notice. As preparing notice is a common process used by several objects, it is put as a generic sub process whose parameters are issuing object and the result to be stored.

```
FROM        : DOAA_INTERFACE.
TO          : SELF.
msg_id      : prepare_reg_notice ().
Action      : do generic sub_process make_notice
              {FROM : $FROM, res_message : reg_notice}
result      : reg_notice <NOTICE>.
```

C.2.2 Comments specification

Finally, any text beginning with a # character upto the end-of-line is considered a comment in all specifications.

References

[Kor92] Abraham Silberscartz Korth. *Database concepts and Systems.* McGraw Hill, 1992.

[ORA92] ORACLE. *ORACLE7 Server Applications Developers guide*, Dec 1992.

[ORA93] ORACLE. *ORACLE7 Server SQL Language Reference Manual*, Dec 1993.

[ORA94] ORACLE. *PL/SQL User's Guide and Reference*, Dec 1994.

[R+88] James Rumbaugh et al. Relational database design using an object-oriented methodology. *Communications of the ACM*, 31(4):414–427, Apr 1988.

[R+90] James Rumbaugh et al. An object-oriented relational database. *Communications of the ACM*, 33:99–109, Nov 1990.

[R+91] James Rumbaugh et al. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

[Raf96] D. Rafee. Translating message centric oo specification to c++. Master's thesis, IIT-Kanpur, Mar 1996.

[Sar96] Burgula Ramanjuneya Sarma. A message based specification system for object oriented software construction. Master's thesis, IIT-Kanpur, Feb 1996.

[Sur95] Surber. Unnatural acts: object applications talking to relational databses. *RothWell International*, May 1995.